Java Object-Oriented Programming Concepts

1. Encapsulation

Encapsulation is the bundling of data (variables) and methods that operate on that data within a single unit (class), while restricting direct access to some components.

Key Features:

- **Data Hiding**: Private variables cannot be accessed directly from outside the class
- **Controlled Access**: Public methods (getters/setters) provide controlled access to private data
- **Security**: Prevents unauthorized access and modification of data

Example:

java

```java
public class Student {
    private String name;        // Private data
    private int age;            // Private data

    // Public getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Public setter methods with validation
    public void setName(String name) {
        if (name != null && !name.trim().isEmpty()) {
            this.name = name;
        }
    }

    public void setAge(int age) {
        if (age > 0 && age < 120) {
            this.age = age;
        }
    }
}
```

2. Abstraction

Abstraction hides implementation details and shows only essential features of an object to the user.

Types of Abstraction:

1. **Abstract Classes**: Cannot be instantiated, may contain abstract and concrete methods
2. **Interfaces**: Pure abstraction, only method signatures (until Java 8)

Abstract Class Example:

java

```java
abstract class Shape {
    protected String color;

    // Concrete method
    public void setColor(String color) {
        this.color = color;
    }

    // Abstract method - must be implemented by subclasses
    public abstract double calculateArea();
    public abstract void draw();
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle with area: " + calculateArea());
    }
}
```

Interface Example:

java

```java
interface Drawable {
    void draw();                // Abstract method (implicitly public abstract)
    default void print() {      // Default method (Java 8+)
        System.out.println("Printing...");
    }
    static void info() {        // Static method (Java 8+)
        System.out.println("Drawable interface");
    }
}

class Rectangle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing rectangle");
    }
}
```

3. Inheritance

Inheritance allows a class to inherit properties and methods from another class, promoting code reusability.

Types:

- **Single Inheritance**: One class extends another class
- **Multilevel Inheritance**: A class extends a class that extends another class
- **Hierarchical Inheritance**: Multiple classes extend the same parent class

Example:

java

```java
// Parent class (Superclass)
class Vehicle {
    protected String brand;
    protected int speed;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void start() {
        System.out.println(brand + " is starting...");
    }

    public void stop() {
        System.out.println(brand + " is stopping...");
    }
}

// Child class (Subclass)
class Car extends Vehicle {
    private int doors;

    public Car(String brand, int doors) {
        super(brand);          // Call parent constructor
        this.doors = doors;
    }

    public void honk() {
        System.out.println(brand + " car is honking!");
    }

    @Override
    public void start() {
        System.out.println("Car " + brand + " is starting with key...");
    }
}
```

4. Multiple Inheritance

Java doesn't support multiple inheritance of classes but supports it through interfaces.

Why Multiple Inheritance of Classes is Not Allowed:

- **Diamond Problem**: Ambiguity when two parent classes have the same method
- **Complexity**: Makes the language more complex

Multiple Inheritance through Interfaces:

java

```java
interface Flyable {
    void fly();
    default void takeOff() {
        System.out.println("Taking off...");
    }
}

interface Swimmable {
    void swim();
    default void dive() {
        System.out.println("Diving...");
    }
}

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}
```

Diamond Problem with Interfaces (Java 8+ Solution):

java

```java
interface A {
    default void display() {
        System.out.println("Interface A");
    }
}

interface B {
    default void display() {
        System.out.println("Interface B");
    }
}

class C implements A, B {
    @Override
    public void display() {
        A.super.display();     // Explicitly call A's method
        B.super.display();     // Explicitly call B's method
        System.out.println("Class C");
    }
}
```

5. this and super Keywords

this Keyword:

Refers to the current instance of the class.

Uses of this:

1. **Distinguish instance variables from parameters**
2. **Call other constructors in the same class**
3. **Pass current object as parameter**
4. **Return current object**

```java
```

```
class Person {
    private String name;
    private int age;

    public Person() {
        this("Unknown", 0);     // Call parameterized constructor
    }

    public Person(String name, int age) {
        this.name = name;       // Distinguish instance variable from parameter
        this.age = age;
    }

    public Person setName(String name) {
        this.name = name;
        return this;            // Return current object for method chaining
    }

    public void display() {
        System.out.println("Name: " + this.name + ", Age: " + this.age);
    }

    public void compare(Person other) {
        if (this.age > other.age) {
            System.out.println(this.name + " is older");
        }
    }
}
```

super Keyword:

Refers to the immediate parent class instance.

Uses of super:

1. **Call parent class constructor**
2. **Access parent class methods**
3. **Access parent class variables**

```
java
```

```
class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    private String breed;

    public Dog(String name, String breed) {
        super(name);          // Call parent constructor
        this.breed = breed;
    }

    @Override
    public void makeSound() {
        super.makeSound();    // Call parent method
        System.out.println("Dog barks");
    }

    public void display() {
        System.out.println("Name: " + super.name + ", Breed: " + this.breed);
    }
}
```

6. Access Modifiers

Control the visibility and accessibility of classes, methods, and variables.

Types:

1. **public**: Accessible from anywhere
2. **protected**: Accessible within package and subclasses
3. **default** (package-private): Accessible within the same package
4. **private**: Accessible only within the same class

Access Modifier Table:

Modifier	Same Class	Same Package	Subclass	Different Package
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

Example:

java

```java
public class AccessExample {
    public int publicVar = 10;       // Accessible everywhere
    protected int protectedVar = 20;   // Accessible in package and subclasses
    int defaultVar = 30;             // Accessible in same package
    private int privateVar = 40;     // Accessible only in this class

    public void publicMethod() { }
    protected void protectedMethod() { }
    void defaultMethod() { }
    private void privateMethod() { }

    // Only private methods can access private variables directly
    public int getPrivateVar() {
        return privateVar;
    }
}
```

7. Dynamic Method Dispatch (Runtime Polymorphism)

The process of calling an overridden method at runtime rather than compile time.

Key Concepts:

- **Method Overriding**: Subclass provides specific implementation of parent class method
- **Late Binding**: Method to be called is determined at runtime
- **Virtual Methods**: All non-static, non-final, non-private methods in Java are virtual

Example:

java

```java
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }

    public void sleep() {
        System.out.println("Animal sleeps");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class DynamicDispatchExample {
    public static void main(String[] args) {
        Animal animal;          // Reference variable of type Animal

        animal = new Dog();     // Object of Dog
        animal.makeSound();     // Calls Dog's makeSound() - Dynamic dispatch

        animal = new Cat();     // Object of Cat
        animal.makeSound();     // Calls Cat's makeSound() - Dynamic dispatch

        // The method called depends on the actual object type, not reference type
    }
}
```

Rules for Dynamic Method Dispatch:

1. **Reference type** determines which methods can be called
2. **Object type** determines which implementation is executed
3. **Only overridden methods** participate in dynamic dispatch
4. **Static, private, and final methods** use static binding

Advanced Example:

java

```java
class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }

    public void calculateArea() {
        System.out.println("Calculating area of shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }

    @Override
    public void calculateArea() {
        System.out.println("Area = π × radius²");
    }

    public void roll() {
        System.out.println("Circle is rolling");
    }
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape[] shapes = {
            new Circle(),
            new Shape(),
            new Circle()
        };

        for (Shape shape : shapes) {
            shape.draw();          // Dynamic dispatch
            shape.calculateArea();  // Dynamic dispatch
            // shape.roll();       // Compilation error - roll() not in Shape
        }
    }
}
```

Key Points to Remember:

1. **Encapsulation** provides data security and controlled access
2. **Abstraction** hides complexity and shows only essential features
3. **Inheritance** promotes code reusability and establishes IS-A relationship
4. **Multiple inheritance** is achieved through interfaces in Java
5. **this** refers to current object, **super** refers to parent class
6. **Access modifiers** control visibility and encapsulation
7. **Dynamic method dispatch** enables runtime polymorphism and flexible code design

These concepts work together to make Java a powerful object-oriented programming language that promotes modularity, reusability, and maintainability.